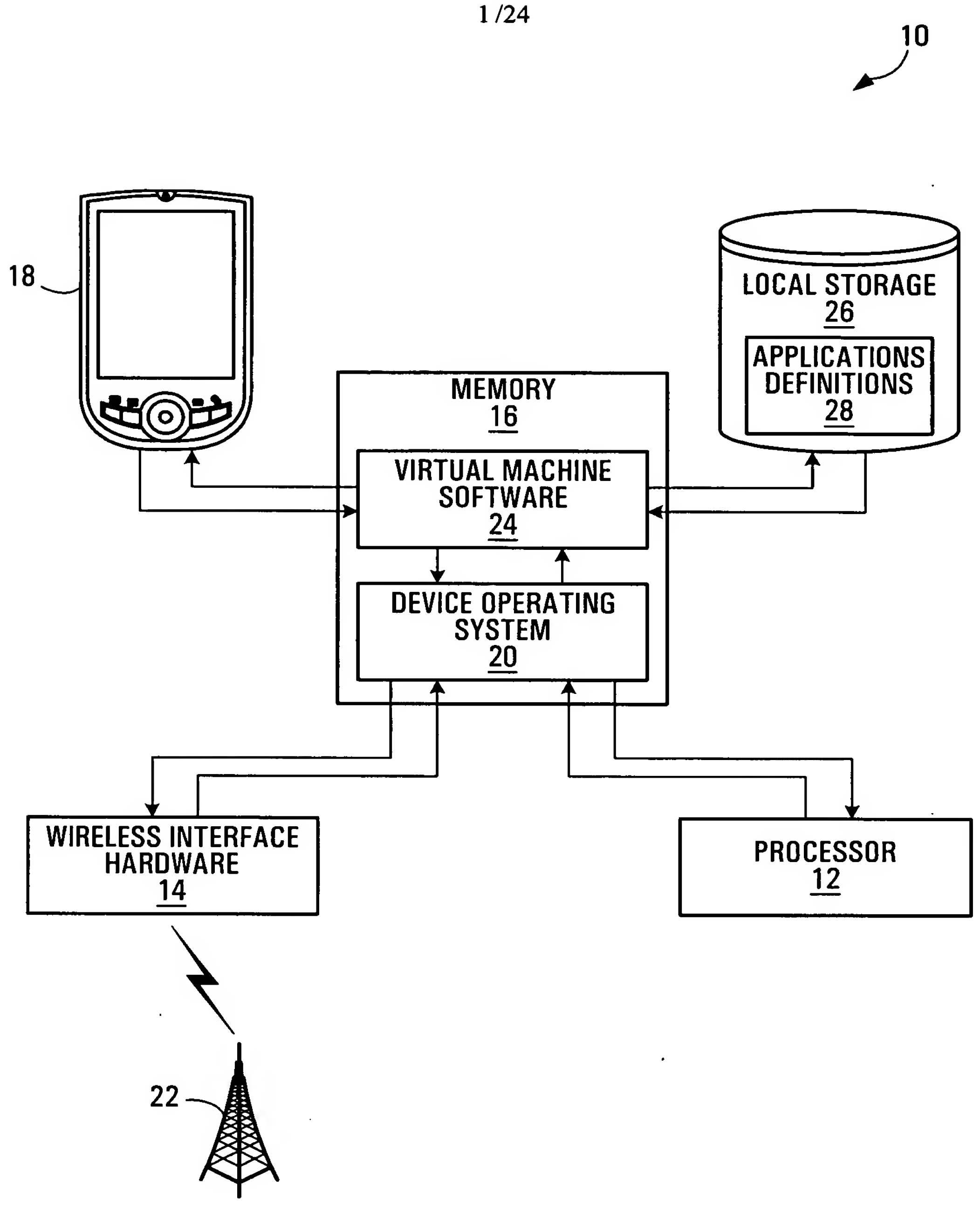
10
18
MEMORY
16
VIRTUAL MACHINE
SOFTWARE
24
DEVICE OPERATING
SYSTEM
20
LOCAL STORAGE
26
APPLICATIONS
DEFINITIONS
28
WIRELESS INTERFACE
HARDWARE
14
PROCESSOR
12
22

FIG. 1

10/537430

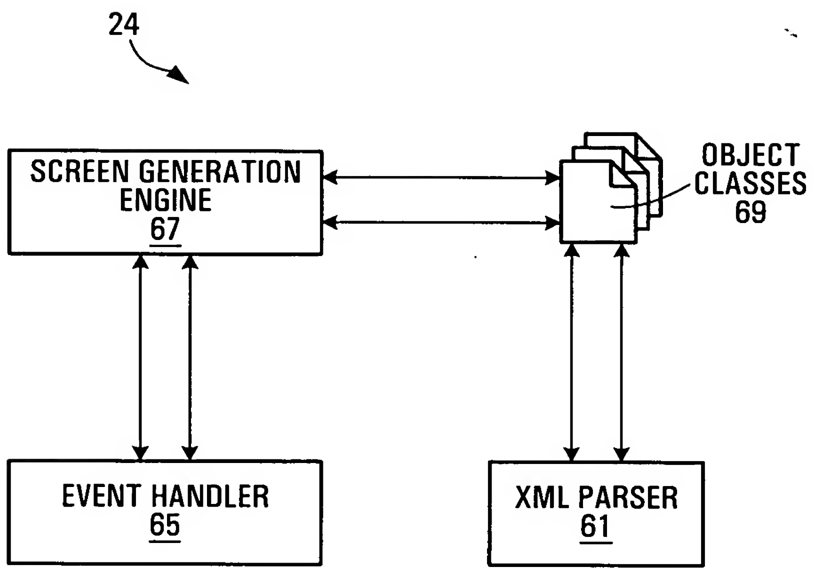

FIG. 2

10/537430

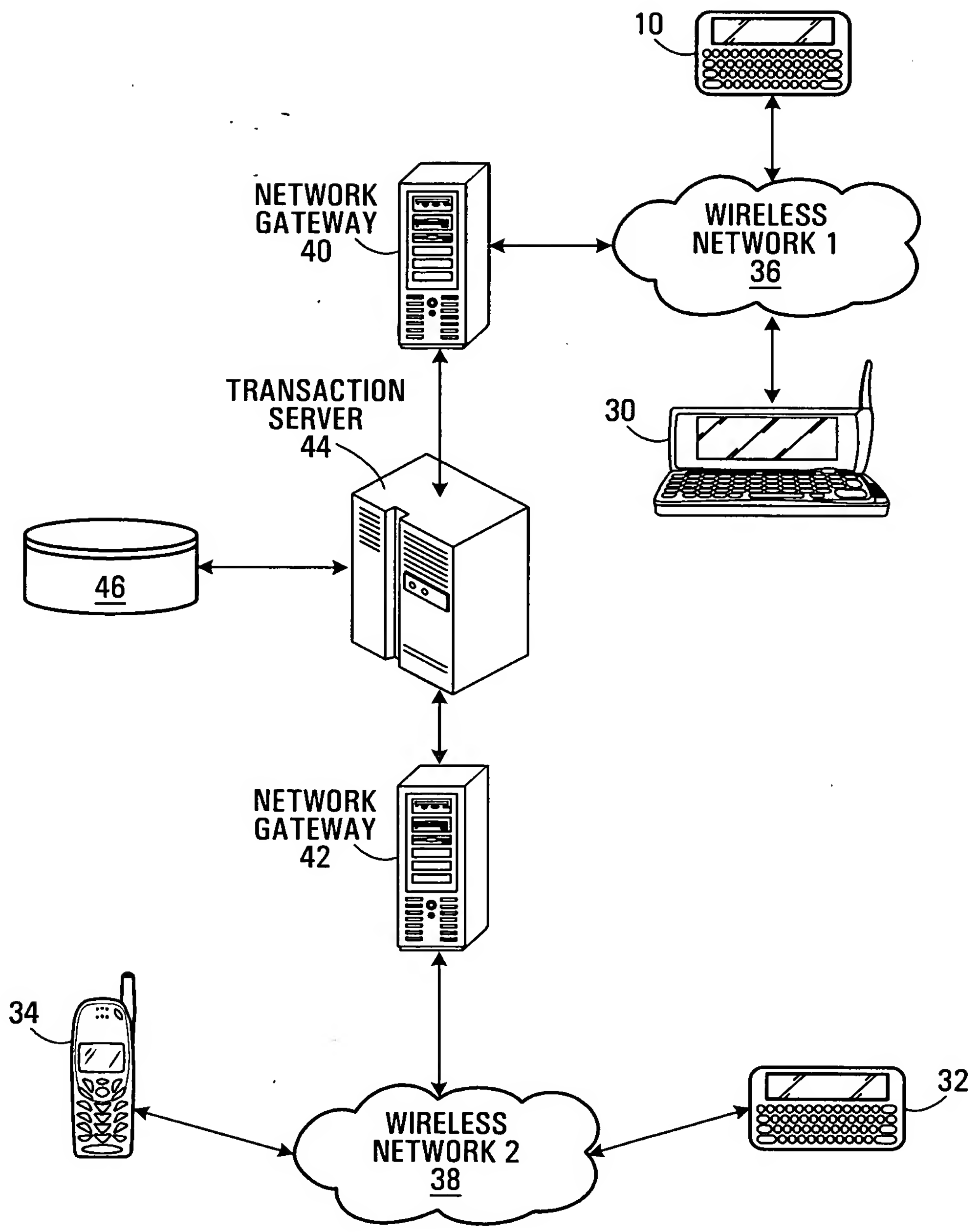

FIG. 3

10/537430

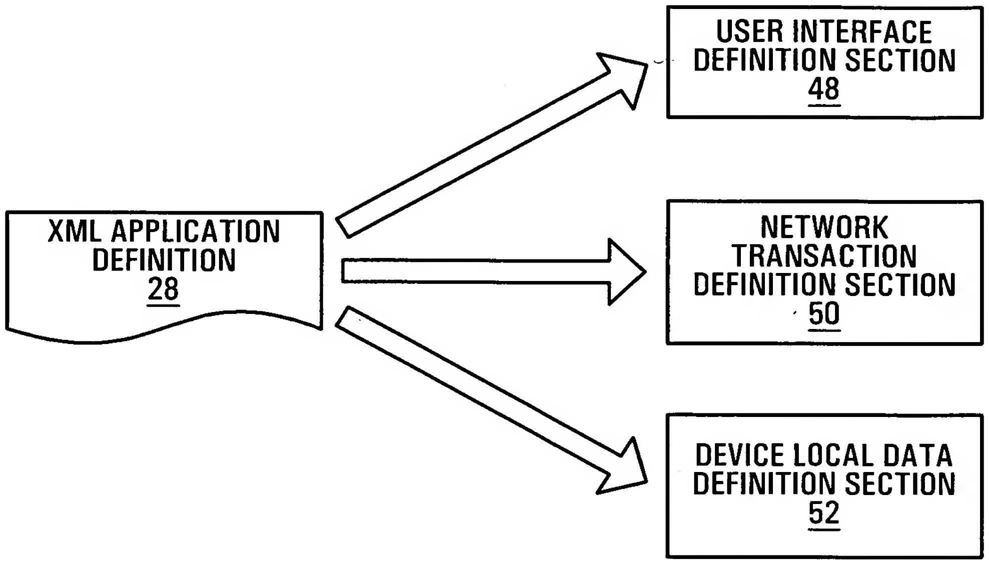
XML APPLICATION DEFINITION 28
USER INTERFACE DEFINITION SECTION 48
NETWORK TRANSACTION DEFINITION SECTION 50
DEVICE LOCAL DATA DEFINITION SECTION 52

FIG. 4

```
<ARML>
     <SCREEN>
          <MENU>
               <MENUITEM>
                    <EVENTS>
                         <ACTION>...</ACTION>
                    <EVENTS>
               </MENUITEM>
          </MENU>
          <BUTTONS>
                      (button definitions)
          </BUTTONS>
          <TEXTITEMS>
                      (textitems definitions)
          </TEXTITEMS>
          <EDITBOXES>
                      (editboxes definitions)
          </EDITBOXES>
          <CHOICEITEMS>
                      (choiceitems definitions)
          </CHOICEITEMS>
          <MESSAGEBOXES>
                      (messageboxes definitions)
          </MESSAGEBOXES>
          <IMAGES>
                      (images definitions)
          </IMAGES>
          <LISTBOXES>
                      (listboxes definitions)
          </LISTBOXES>
          <CHECKBOXES>
                      (checkboxes definitions)
          </CHECKBOXES>
     </SCREEN>
</ARML>
```

FIG. 5

10/537430

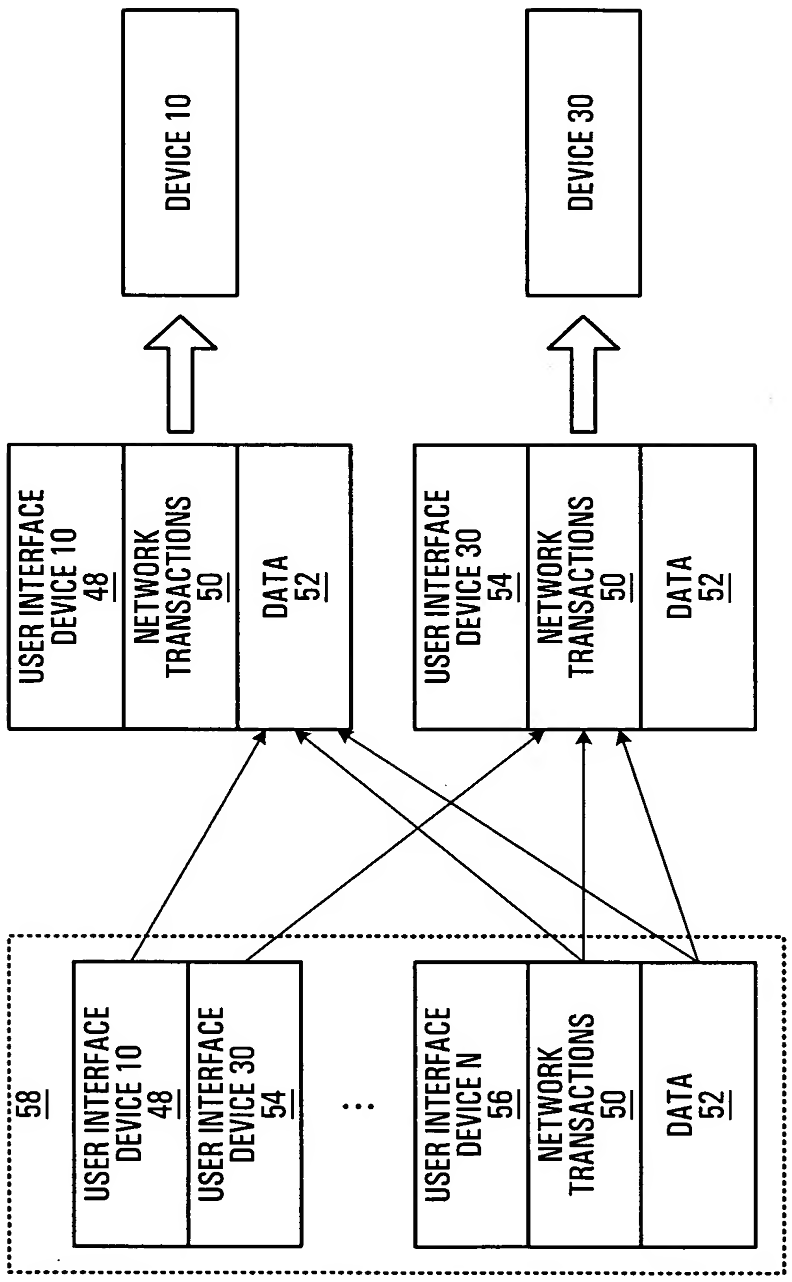
DEVICE 10
DEVICE 30
USER INTERFACE DEVICE 10 48
NETWORK TRANSACTIONS 50
DATA 52
USER INTERFACE DEVICE 30 54
NETWORK TRANSACTIONS 50
DATA 52
58
USER INTERFACE DEVICE 10 48
USER INTERFACE DEVICE 30 54
...
USER INTERFACE DEVICE N 56
NETWORK TRANSACTIONS 50
DATA 52

FIG. 6

10/537430

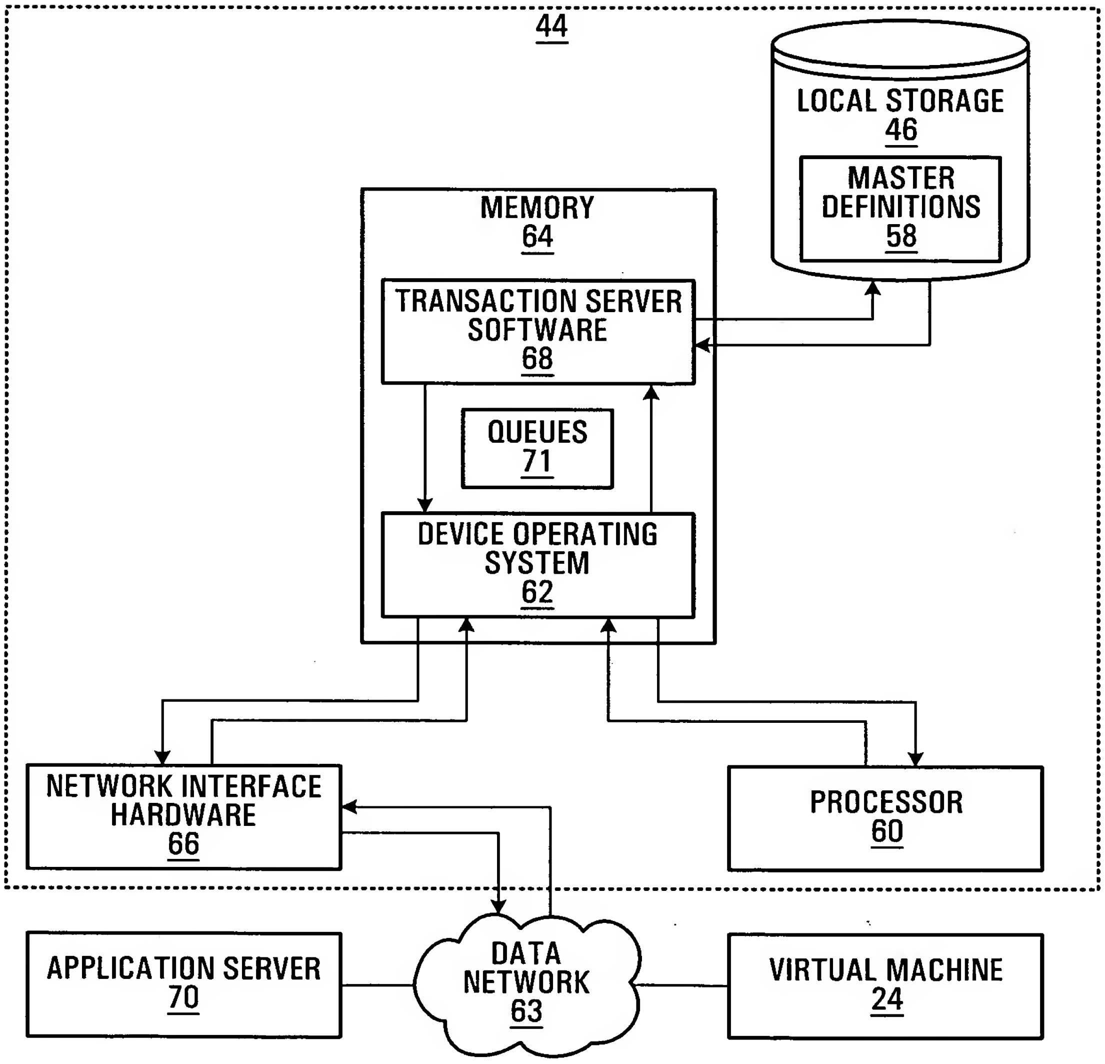

FIG. 7

```
private void _Send(int applicationID, int mobileID, String
                        message,
                       int messageType)
                            {
                            try
                             {
        // Insert message into the application queue
                             ...

   // Lookup delivery type and push details for application
String sql = "SELECT LNGDELIVERYTYPE, TXTDELIVERYDETAILS, " +
     "FROM TBLAPPLICATIONS WHERE LNGID = " + applicationID;

              IDataReader reader = ExecuteQuery(sql);
                     if (!reader.Read())
                  throw new Exception("...");

     int deliveryType = (int)reader["LNGDELIVERYTYPE"];
                   String deliveryDetails =
             (String)reader["TXTDELIVERYDETAILS"];

                 if (deliveryType != POLL_HTTP)
                              {
        // This application uses a push delivery type
  IAIRIXEnterpriseWakeup entWakeup = (IAIRIXEnterpriseWakeup)
                Marshal.BindToMoniker("queue:/new:" +

"Nextair.AIRIX.Server.Enterprise.Router.AIRIXEnterpriseWakeup");
        entWakeup.Wakeup(applicationID, deliveryType,
                    deliveryDetails);
                              }
                            }
```

FIG. 8

```
                    public void Retry()
                              {
  // Select all push-enabled applications that have expired
                       queued messages
       string sql = "SELECT DISTINCT LNGAPPLICATIONID,
                    LNGDELIVERYTYPE, " +
                          "TXTDELIVERYDETAILS " +
            "FROM TBLAPPLICATIONS A, TBLAPPLICATIONQUEUE Q " +
               "WHERE A.LNGDELIVERYTYPE <> " + POLL_HTTP +
                  "AND Q.LNGAPPLICATIONID = A.LNGID " +
                     "Q.DTMQUEUED < " + expiryDate;

         // Get a disconnected list of apps to retry
          NextairDisconnectedDataProvider dp = new
            NextairDisconnectedDataProvider();
             DataSet ds = dp.ExecuteQuery(sql);
         foreach (DataRow row in ds.Tables[0].Rows)
                               {
                int appID = (int)row["LNGID"];
        int deliveryType = (int)row["LNGDELIVERYTYPE"];
  string deliveryDetails = (string)row["TXTDELIVERYDETAILS"];

  // Call the Wakeup method for this application asyncronously
  IAIRIXEnterpriseWakeup entWakeup = (IAIRIXEnterpriseWakeup)
             Marshal.BindToMoniker("queue:/new:" +

"Nextair.AIRIX.Server.Enterprise.Router.AIRIXEnterpriseWakeup");
    entWakeup.Wakeup(appID, deliveryType, deliveryDetails);
                               }
                              }
```

FIG. 9

```
public class AIRIXLockManager
{
private static Object lockSync = new Object();
private static Hashtable locks = new Hashtable();

public static bool ObtainLock(int lockID)
{
// Make sure only one caller can attempt to obtain a lock at
once.
// We don't bother locking per application, since this method
is
// expected to execute extremely quickly.
lock (lockSync)
{
if (!locks.Contains(lockID))
{
// This is the first attempt to lock this lock ID
locks[lockID] = true;
return true;
}

if (locks[lockID])
return false; // this ID is already locked

// Can successfully obtain the lock for this lock ID
locks[lockID] = true;
return true;
}
}

public static void ReleaseLock(int lockID)
{
lock (lockSync)
{
locks[lockID] = false;
}
}
}
```

FIG. 10

```
[InterfaceQueuing]
public interface IAIRIXEnterpriseWakeup
{
void Wakeup(int appID, int deliveryType, String
deliveryDetails);
}

public class AIRIXEnterpriseWakeup :
NextairDatabase, IAIRIXEnterpriseWakeup
{
private bool _clustered = false;
private String _lockProvider = String.Empty;

public void OnConstruct(String constructString)
{
clustered = ...; // read config from config
if (clustered)
_lockProvider = ...; // read lock server location from
config
}

private bool _obtainLock(int appID)
{
if (_clustered)
return RemotingServer.ObtainLock(appID); // call remoting
server
else
return AIRIXLockManager.ObtainLock(appID); // obtain local
lock
}

private void _releaseLock(int appID)
{
if (_clustered)
RemotingServer.ReleaseLock(appID); // call remoting server
else
AIRIXLockManager.ReleaseLock(appID); // release local lock
}

public void Wakeup(int appID, int deliveryType,
String deliveryDetails)
{
if (_obtainLock(appID))
{
try
{
// Obtain a disconnected list of queued messages, ordered
// oldest -> newest.
DataSet messages = RetrieveQueuedMessages(appID);
```

FIG. 11A

```
// Loop through each queued message and attempt to push
it
foreach (DataRow msg in messages.Tables[0].Rows)
{
AIRIXEnterprisePushBase pushObj = null;
try
{
pushObj = _createPushComponent(deliveryType);
if (pushObj ==  null)
... // unable to create push component for delivery
type

// Synchronously push this message out
int result = pushObj.Push(appID, (int)msg["LNGID"],
(int)msg["LNGMESSAGETYPEID"],
(String)msg["VARMOBILEID"], deliveryDetails);

if (!AIRIXConstants.Succeeded(result))
... // throw exception so that pushing stops
}
catch (Exception x)
{
... // log this push msg error and break out of push
loop
}
finally
{
NextairServicedComponent.DisposeComponent(pushObj);
}
}
}
finally
{
_releaseLock(appID);
}
}
}
}
```

FIG. 11B

10/537430

```
public interface IAIRIXEnterprisePush
{
// Called by AIRIX to push an application-bound message out
bool AIRIXReceiveData(int appID, string mobileID, string
data);

// Called by AIRIX when a mobile-bound message delivery fails
bool AIRIXDeliveryError(int appID, string mobileID, string
data,
int errorCode, string errorDescription);

// Called by AIRIX when a mobile-bound message is delivered
bool AIRIXDeliveryNotify(int appID, string mobileID, string
data);
}
```

FIG. 12

```
public class AIRIXEnterprisePushBase : NextairDatabase
{
    protected int moveQueueToLog(int queueID)
    {
        // Move the specified message from the Application Queue
        // to the Application Log, and return an appropriate
        // AIRIXConstants result.
        ...
    }

    public int Push(int appID, int queueID, String message,
    int messageType, String mobileID, String deliveryDetails)
    {
        IAIRIXEnterprisePush pushClient = null;
        try
        {
            // Move the message from the queue to the log first. It
            will be
            // rolled back if the PUSH fails.
            int result = moveQueueToLog(queueID);
            if (!AIRIXConstants.Succeeded(result))
                ... // abort the transaction and return error

            // Create an instance of IAIRIXEnterprisePush.
            // This logic is left up the the child class, since this
            process
            // can differ depending on the type of communication used.
            pushClient = createPushClient(appID, deliveryDetails);

            if (pushClient == null)
                throw new Exception("Invalid interface reference.");

            // Push the message out using the retrieved interface
            bool success = false;
            switch (messageType)
            {
                case MessageTypes.APPLICATION_DATA:
                    success = pushClient.AIRIXReceiveData(appID,
                        mobileID, message);
                    break;

                case MessageTypes.DELIVERY_CONFIRMATION:
                    success = pushClient.AIRIXDeliveryNotify(appID,
                        mobileID, message);
                    break;

                case MessageTypes.FAILURE_NOTIFICATION:
                    int errorCode = getErrorCode(message);
                    int errorMsg  = getErrorMsg(message);
                    success = pushClient.AIRIXDeliveryError(appID,
                        mobileID, message, errorCode, errorMsg);
                    break;
```

FIG. 13A

10/537430

```
default:
throw new Exception("Invalid message type: " +
messageType);
}
if (!success)
{
// Log error or warning and exit...
_SetAbort();
return AIRIXConstants.ENTERPRISE_PUSH_FAILED;
}

return AIRIXConstants.SUCCESS;
}
catch (Exception x)
{
// Log error...
_SetAbort();
return AIRIXConstants.ENTERPRISE_PUSH_UNKNOWN_ERROR;
}
finally
{
if (pushClient != null)
disposePushClient(pushClient);
_SetComplete();
}
}

// Abstract method that children will implement to do the work
// of pushing the message to an application.
protected abstract IAIRIXEnterprisePush createPushClient(int
appID,
String deliveryDetails);

// Overridable method that children can implement to provide
// component specific cleanup of the IAIRIXEnterprisePush
client
// created via the createPushClient method.
protected void disposePushClient(IAIRIXEnterprisePush
pushClient)
{
// Base class implementation does nothing.
// Children can optionally override this to perform explicit
// cleaning up of the previously created push client
component.
}
}
```

FIG. 13B

```
                    [
uuid(EF2795BE-3874-4ACF-A087-8113FB791211),
              version(1.0),
helpstring("IAIRIXEnterprisePush Library")
                    ]
      library IAIRIXEnterprisePush
                    {
        importlib("STDOLE2.TLB");

                     [
 uuid(E7C20DA3-6820-4D3D-8E5C-A8BE61BFDFFE),
               version(1.0),
                   dual,
               oleautomation
                     ]
 interface IAIRIXEnterprisePush: IDispatch
                     {
              [id(0x00000001)]
HRESULT _stdcall AIRIXReceiveData([in] long appID,
        [in] BSTR mobileID, [in] BSTR data,
       [out, retval] VARIANT_BOOL * Result);

              [id(0x00000002)]
HRESULT _stdcall AIRIXDeliveryError([in] long appID,
[in] BSTR mobileID, [in] BSTR data, [in] long errorCode,
          [in] BSTR errorDescription,
       [out, retval] VARIANT_BOOL * Result);

              [id(0x00000003)]
HRESULT _stdcall AIRIXDeliveryNotify([in] long appID,
        [in] BSTR mobileID, [in] BSTR data,
       [out, retval] VARIANT_BOOL * Result);
                    };
                   };
```

FIG. 14

```
protected IAIRIXEnterprisePush createPushClient(int appID,
                        String deliveryDetails)
                        {
        // Get type/interface information from COM object
      Type type = Type.GetTypeFromProgID(deliveryDetails);
                        if (type == null)
        throw new Exception("Unable to create COM object: " +
                                        deliveryDetails);

              // Create an instance of the COM object
        Object instance = Activator.CreateInstance(type);
              if (!(instance is IAIRIXEnterprisePush))
      throw new Exception("COM object does not implement the "
                                        +
                        "IAIRIXEnterprisePush interface.");

              return (IAIRIXEnterprisePush)instance;
                        }
```

FIG. 15

10/537430

```
// Called by the ATS to deliver application-bound messages
bool AIRIXReceiveData(int appID, String mobileID, String data);

// Called by the ATS when a mobile-bound message delivery fails
bool AIRIXDeliveryError(int appID, String mobileID, String data,
        int errorCode, String errorDescription);

// Called by the ATS when a mobile-bound message is
        // successfully delivered.
bool AIRIXDeliveryNotify(int appID, String mobileID, String
        data);
```

FIG. 16

```
// Import the WSDL definition into a CodeDom namespace
ServiceDescriptionImporter imp = new
ServiceDescriptionImporter();
DiscoveryProtocol dcp = new DiscoveryProtocol();
dcp.DiscoverAny(wsdlLocation);
dcp.ResolveAll();
foreach (object o in dcp.Documents.Values)
{
if (o is ServiceDescription)
imp.AddServiceDescription((ServiceDescription)o, null, null);
if (o is XmlSchema)
imp.Schemas.Add((XmlSchema)o);
}
CodeNamespace ns = new CodeNamespace(
"Nextair.AIRIX.Server.Enterprise.Push.WSDL");
imp.ProtocolName = "Soap";
imp.Import(ns, null);

// Verify that all classes in the namespace have the proper
signature
bool m1 = false, m2 = false, m3 = false;
foreach (CodeTypeDeclaration t in ns.Types)
{
if (t.IsClass && t.Name == typeName)
{
foreach (CodeTypeMember m in t.Members)
{
if (m.Name == "AIRIXReceiveData") m1 = true;
else if (m.Name == "AIRIXDeliveryError") m2 = true;
else if (m.Name == "AIRIXDeliveryNotify") m3 = true;
}
t.BaseTypes.Add("Nextair.AIRIX.Server.Enterprise." +
"Push.IAIRIXEnterprisePush");
break;
}
}
if (!(m1 && m2 && m3))
throw new Exception("Incomplete interface definition.");
```

FIG. 17A

```
// Generate source code from the imported web service
CSharpCodeProvider provider = new CsharpCodeProvider();
ICodeGenerator gen = provider.CreateGenerator();
StringBuilder sb = new StringBuilder();
StringWriter sw = new StringWriter(sb);
gen.GenerateCodeFromNamespace(ns, sw, null);
string sourceCode = sb.ToString();
sb.Close();

// Compile the proxy assembly
CompilerParameters cp = new CompilerParameters();
cp.GenerateExecutable = false;
cp.GenerateInMemory = false;
cp.IncludeDebugInformation = false;
cp.ReferencedAssemblies.Add("System.dll");
cp.ReferencedAssemblies.Add("System.Xml.dll");
cp.ReferencedAssemblies.Add("System.Web.Services.dll");
cp.ReferencedAssemblies.Add("System.Data.dll");
cp.ReferencedAssemblies.Add(
typeof(IAIRIXEnterprisePush).Assembly.Location);
cp.OutputAssembly = proxyDir + applicationID.ToString() + ".dll";
ICodeCompiler compiler = cscp.CreateCompiler();
CompilerResults results;
results = compiler.CompileAssemblyFromSource(cp, sourceCode);
if (results.Errors.Count > 0)
throw new Exception("Build failed.");

// Cache handle to compiled assembly and add the soap interface
proxy
// type to the static list of cached proxies.
Assembly asm = results.CompiledAssembly;
Type proxyType = asm.GetType(
"Nextair.AIRIX.Server.Enterprise.WSDL." + typeName, true,
true);
cachedProxies[wsdlLocation] = proxyType;
```

FIG. 17B

10/537430

```
protected IAIRIXEnterprisePush createPushClient(
        int appID, String deliveryDetails)
                    {
// Pull service location and name out of delivery details xml
        XmlDocument xml = new XmlDocument();
            xml.LoadXml(deliveryDetails);
                string serviceUrl  =
    xml.DocumentElement.Attributes["Url"].Value;
                string serviceName =
    xml.DocumentElement.Attributes["Name"].Value;

   // If the proxy for this service url is not yet cached,
            // build and cache it now.
        if (!cachedProxies.Contains(serviceUrl))
        _buildSoapProxy(serviceUrl, serviceName);

// Create an instance of the soap proxy for this web service
    Type type = (Type)cachedProxies[serviceUrl];
        IAIRIXEnterprisePush pushClient =
  (IAIRIXEnterprisePush)Activator.CreateInstance(type);
                if (push == null)
   throw new Exception("Unable to create proxy instance.");

                return (pushClient);
                    }
```

FIG. 18

```
public IAIRIXEnterprisePush createPushClient(int appID,
                String deliveryDetails)
                        {
            // Retrieve delivery information
          XmlDocument xml = new XmlDocument();
             xml.LoadXml(deliveryDetails);
                 string serviceName =
    xml.DocumentElement.Attributes["Name"].Value;
             int servicePort = Int32.Parse(
       xml.DocumentElement.Attributes["Port"].Value);
                string serviceLocation =
      xml.DocumentElement.Attributes["Location"].Value;

    // Return a remote instance of IAIRIXEnterprisePush
    return (IAIRIXEnterprisePush)Activator.GetObject(
               typeof(IAIRIXEnterprisePush),
    "tcp://" + serviceLocation + ":" + servicePort + "/" +
                      serviceName);
                        }
```

FIG. 19

10/537430

```
private void RetryTimer(object sender, EventArgs e)
{
    AIRIXEnterpriseRouter router = null;
    try
    {
        router = new AIRIXEnterpriseRouter();
        router.Retry();
    }
    catch (Exception x)
    {
        ...
    }
    finally
    {
        NextairServicedComponent.DisposeComponent(router);
    }
}
```

FIG. 20

```
// Register a tcp channel for listening for remoting requests
TcpChannel channel = new TcpChannel(configuredPortNumber);
ChannelServices.RegisterChannel(channel);

// Expose the AIRIXRemotingLockManager as a singleton type
lockManager = new AIRIXRemotingLockManager();
RemotingServices.Marshal(lockManager, "LockManager");
```

FIG. 21